

To Loom or To Reactive

That is the question

Oleh Dokuka & Andrii Rodionov

You want

You want

- Figure out whether to learn Reactive or not

You want

- Figure out whether to learn Reactive or not
- Figure out how **Loom** is going to affect **Reactive** Landscape

In this talk

In this talk

- Problems of modern app development

In this talk

- Problems of modern app development
- CompletableFutures and Project Reactor

In this talk

- Problems of modern app development
- CompletableFuture and Project Reactor
- Loom

NOT in this talk

NOT in this talk

- Loom internals

NOT in this talk

- Loom internals

Joker<?>

Alan Bateman

Oracle Corporation

**Project loom: Modern scalable
concurrency for the Java platform**



NOT in this talk

- Loom internals
- Project Reactor reimplementation

Joker<?>

Alan Bateman

Oracle Corporation

**Project loom: Modern scalable
concurrency for the Java platform**



NOT in this talk

- Loom internals
- Project Reactor reimplementation

Joker<?>

Alan Bateman

Oracle Corporation

**Project loom: Modern scalable
concurrency for the Java platform**

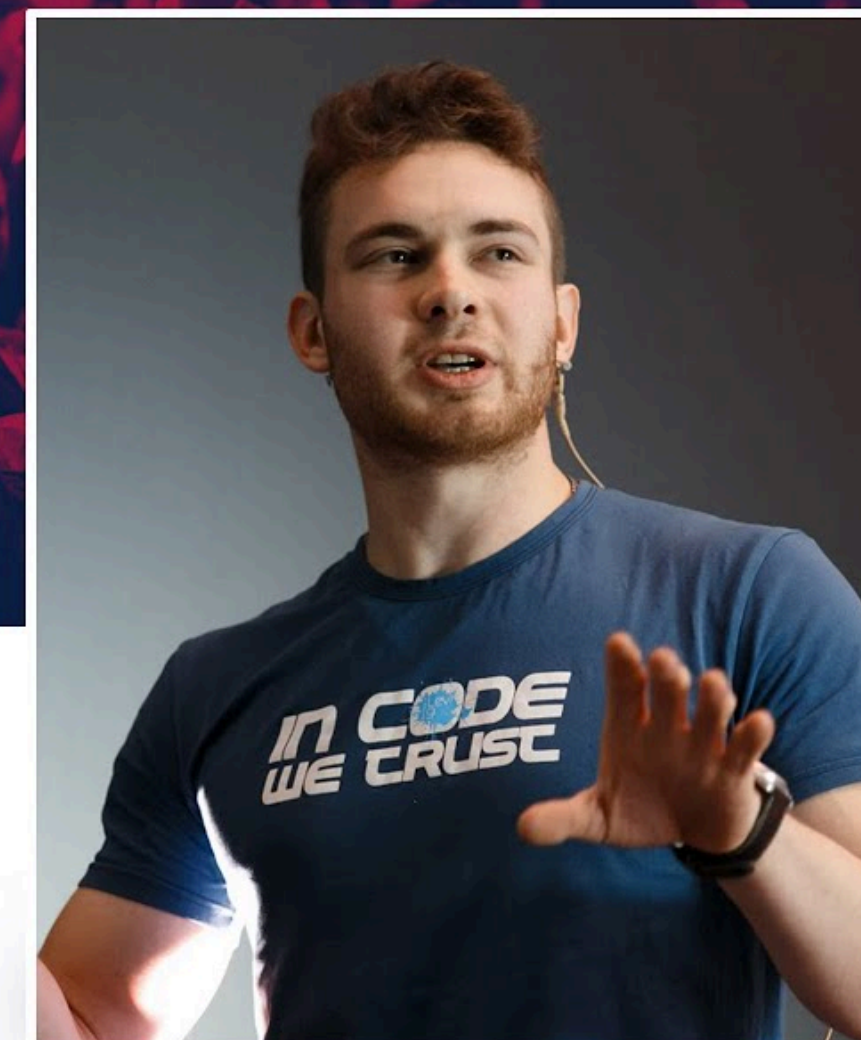


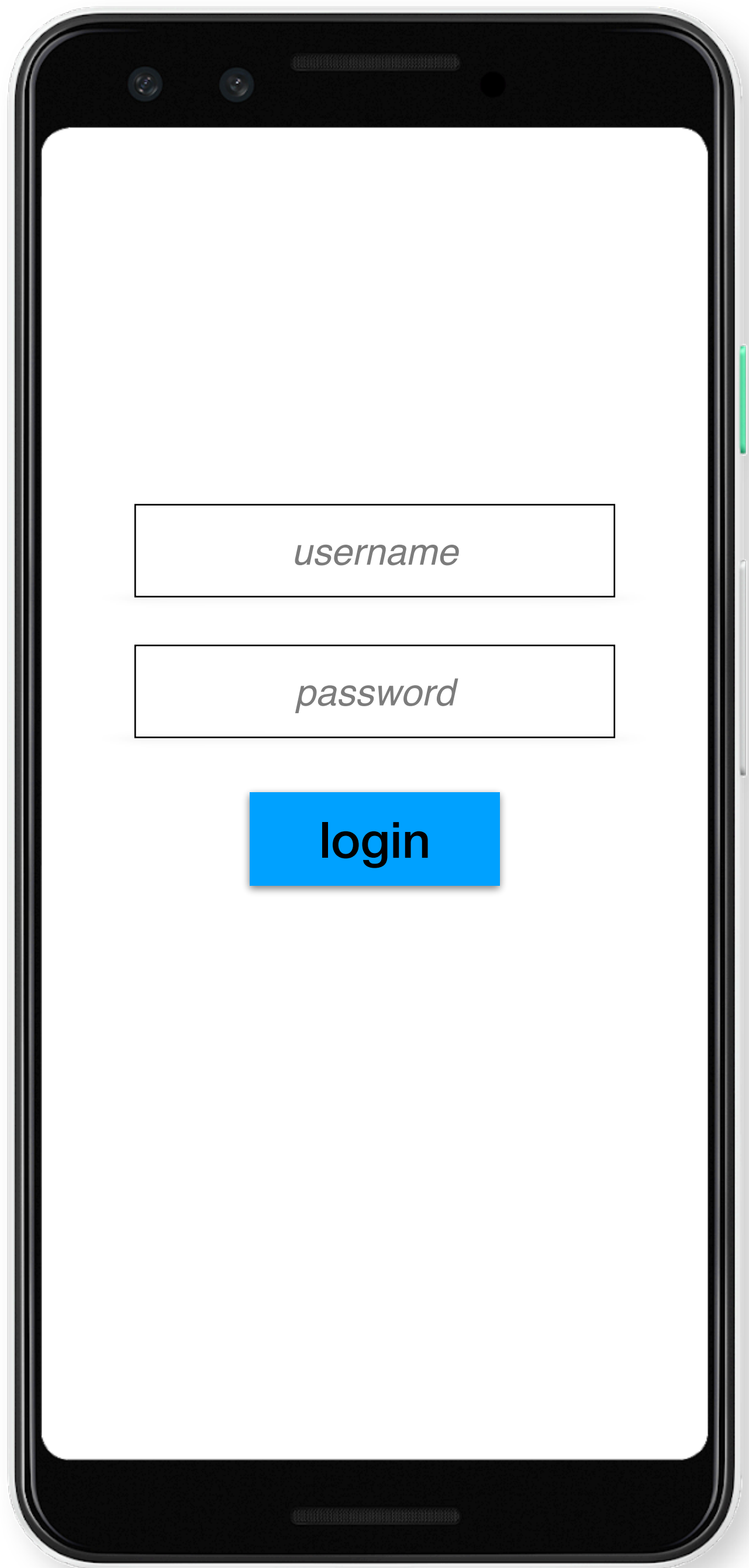
Joker<?> 2018

Олег Докука

Netifi

Реактивный хардкор:
как построить свой
Publisher<?>

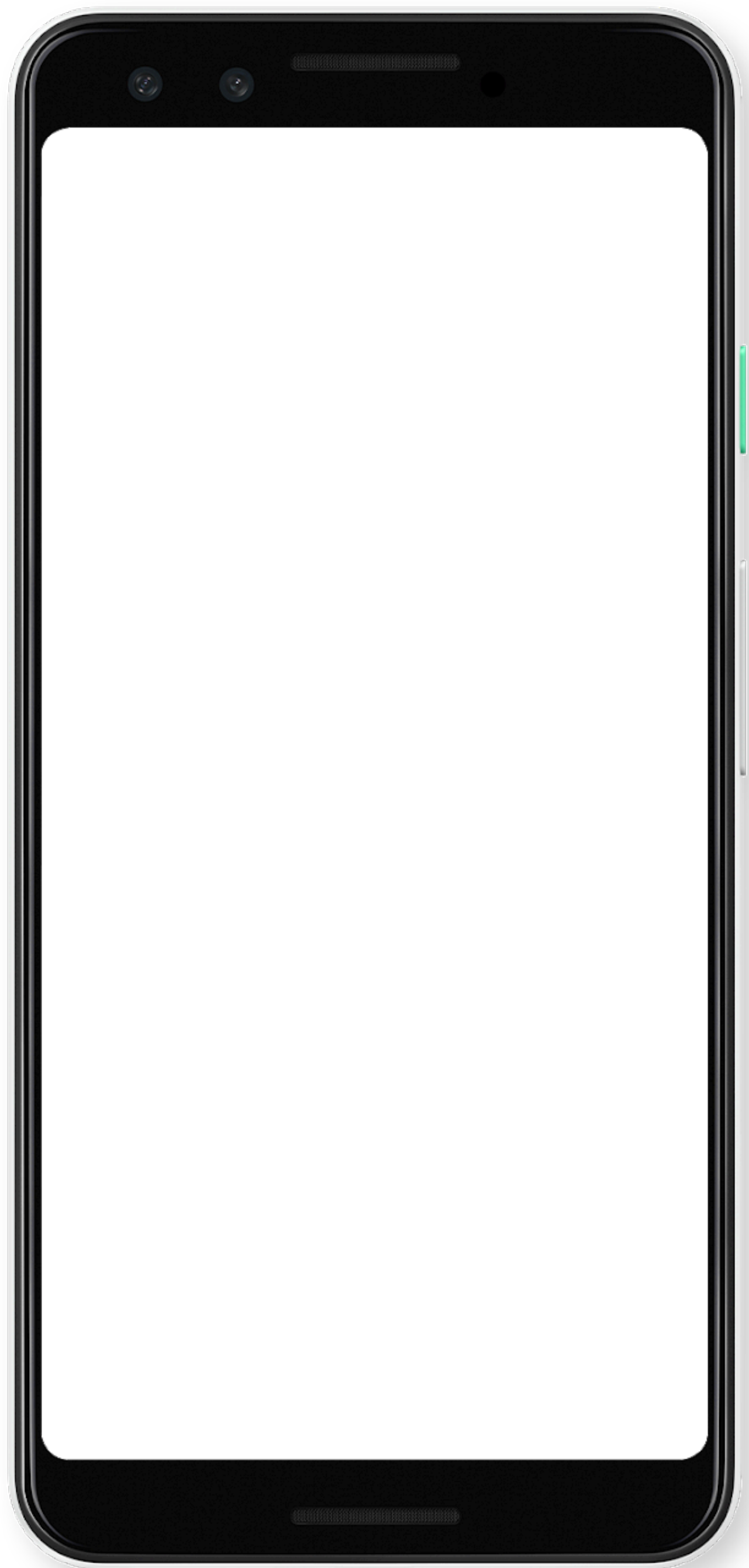


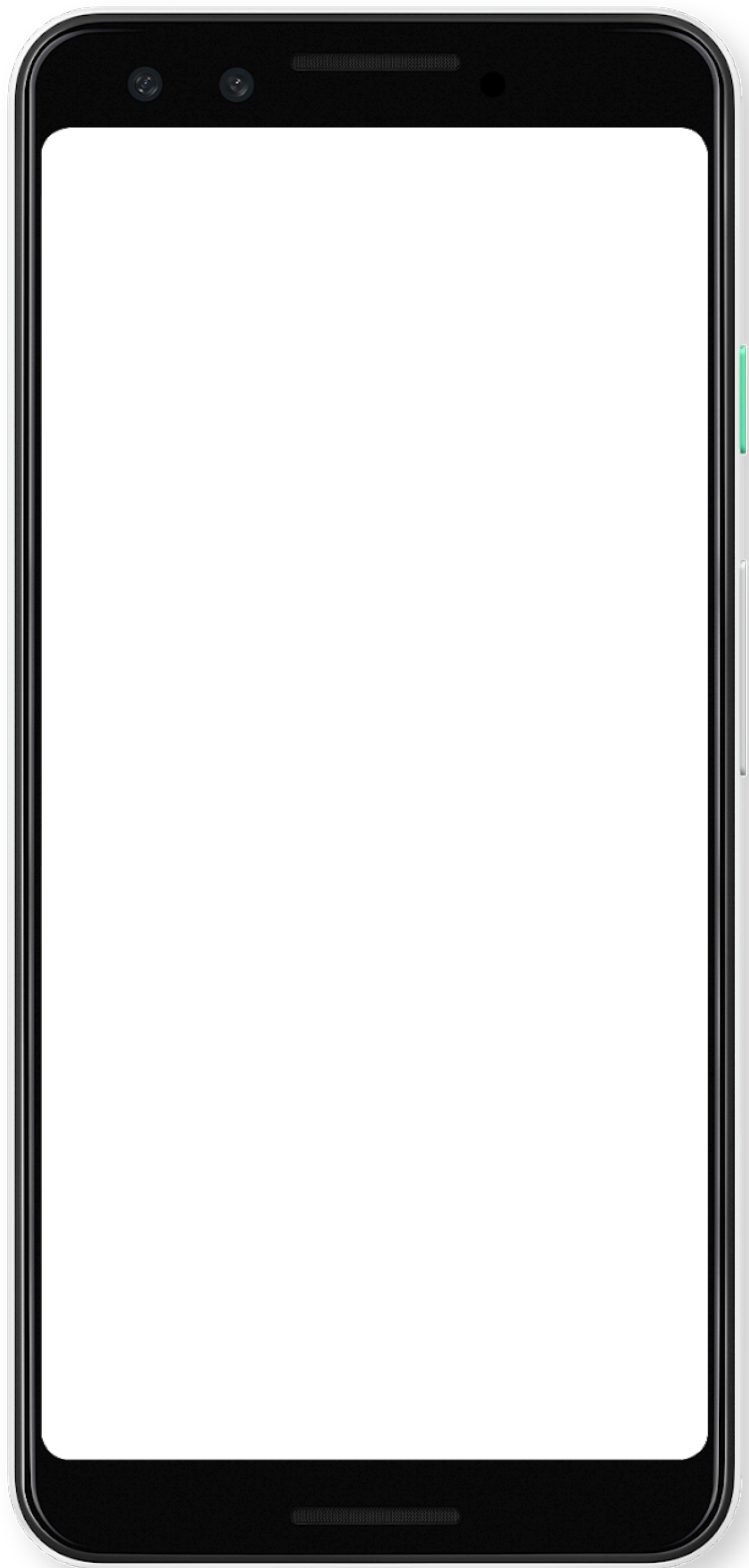


username

password

login

















```
List<SensorInfo> listUsersHouseSensors(  
    String userId,  
    String houseId) {  
    final List<HouseInfo> houses =  
        housesRepository.findByUserId(userId);  
  
    for (HouseInfo house : houses) {  
        if (houseId.equals(house.id())) {  
            return sensorsRepository.findByIds(house.sensorIds());  
        }  
    }  
}
```




```
List<SensorInfo> listUsersHouseSensors(  
    String userId,  
    String houseId) {  
    final List<HouseInfo> houses =  
        housesRepository.findByUserId(userId);  
  
    for (HouseInfo house : houses) {  
        if (houseId.equals(house.id())) {  
            return sensorsRepository.findByIds(house.sensorIds());  
        }  
    }  
}
```












```
interface SensorClient {  
    double temperatureForSensorId(String id);  
}
```



```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
  
for (SensorInfo sensorInfo : sensorInfos) {  
    final double temperature = sensorClient.temperatureForSensorId(sensorInfo.id());  
    final SensorData sensorData = new SensorData(sensorInfo, temperature, Instant.now());  
    datum.add(sensorData);  
}  
  
return datum;
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
  
for (SensorInfo sensorInfo : sensorInfos) {  
    final double temperature = sensorClient.temperatureForSensorId(sensorInfo.id());  
    final SensorData sensorData = new SensorData(sensorInfo, temperature, Instant.now());  
    datum.add(sensorData);  
}  
  
return datum;
```




```

final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());

for (SensorInfo sensorInfo : sensorInfos) {
    final double temperature = sensorClient.temperatureForSensorId(sensorInfo.id());
    final SensorData sensorData = new SensorData(sensorInfo, temperature, Instant.now());
    datum.add(sensorData);
}

return datum;

```




```

final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());

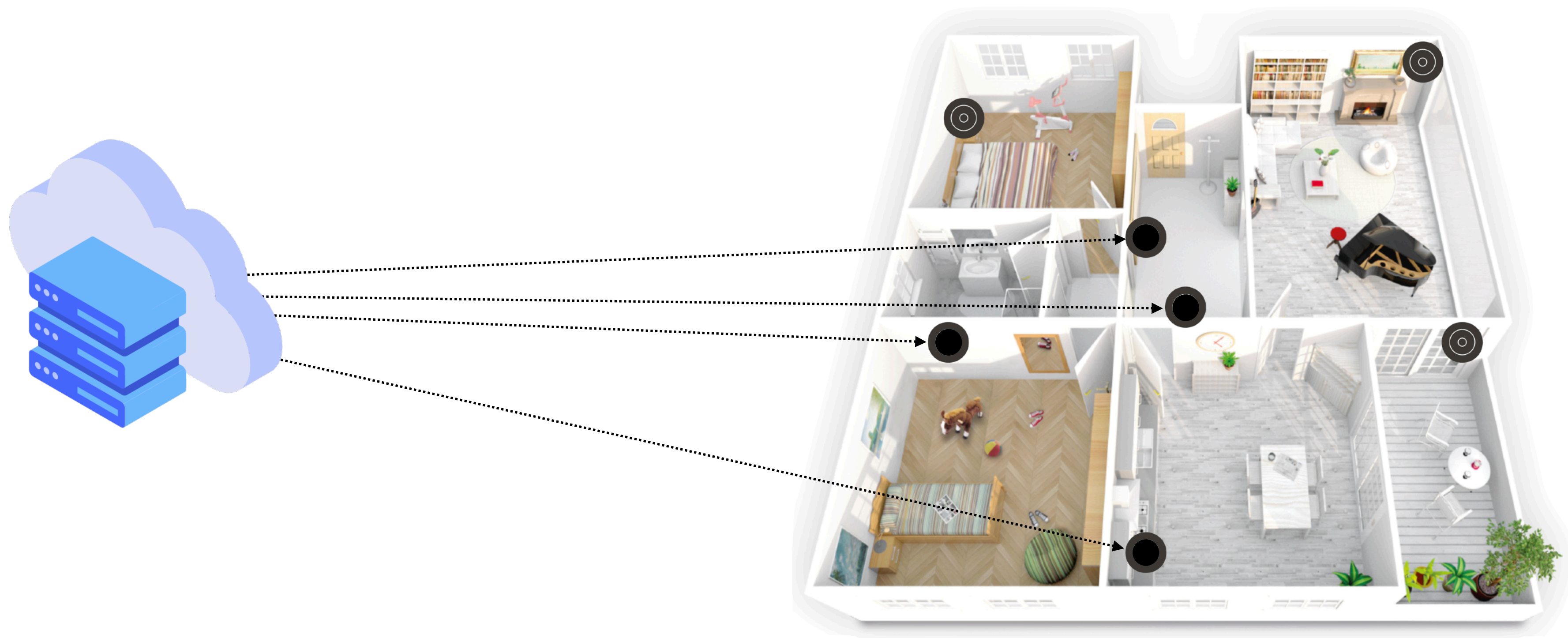
for (SensorInfo sensorInfo : sensorInfos) {
    final double temperature = sensorClient.temperatureForSensorId(sensorInfo.id());
    final SensorData sensorData = new SensorData(sensorInfo, temperature, Instant.now());
    datum.add(sensorData);
}

return datum;

```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
  
for (SensorInfo sensorInfo : sensorInfos) {  
    final double temperature = sensorClient.temperatureForSensorId(sensorInfo.id());  
    final SensorData sensorData = new SensorData(sensorInfo, temperature, Instant.now());  
    datum.add(sensorData);  
}  
  
return datum;
```





**RESPONSE
TIME**





```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();
```

```
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();
```

```
final List<Future<SensorData>> futures = executor.invokeAll(callables);
```

```
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());
```

```
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();  
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();  
final List<Future<SensorData>> futures = executor.invokeAll(callables);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();
```

```
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();
```

```
final List<Future<SensorData>> futures = executor.invokeAll(callables);
```

```
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());
```

```
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```



```

final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);

final List<Callable<SensorData>> callables = new ArrayList<>();
for (SensorInfo sensorInfo : sensorInfos) { ... }

final ExecutorService executor = Executors.newCachedThreadPool();
final List<Future<SensorData>> futures = executor.invokeAll(callables);
for (SensorInfo sensorInfo : sensorInfos) {
    for (Future<SensorData> future : futures) {
        callables.add(() -> {
            double t = sensorClient.temperatureForSensorId(sensorInfo.id());
            SensorData sensorData = new SensorData(...);
            return sensorData;
        });
    }
}

```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();
```

```
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();
```

```
final List<Future<SensorData>> futures = executor.invokeAll(callables);
```

```
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());
```

```
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();  
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();  
final List<Future<SensorData>> futures = executor.invokeAll(callables);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();  
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();  
final List<Future<SensorData>> futures = executor.invokeAll(callables);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```







```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();
```

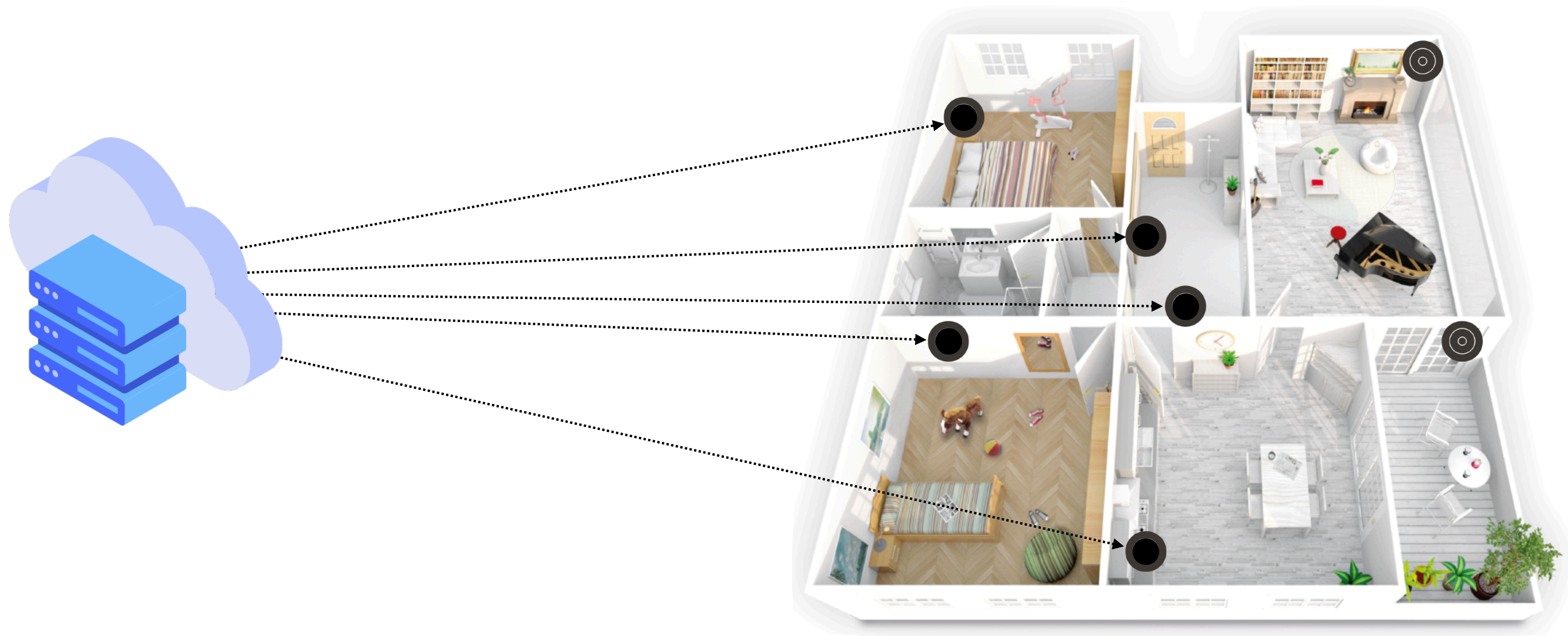
```
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();
```

```
final List<Future<SensorData>> futures = executor.invokeAll(callables);
```

```
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());
```

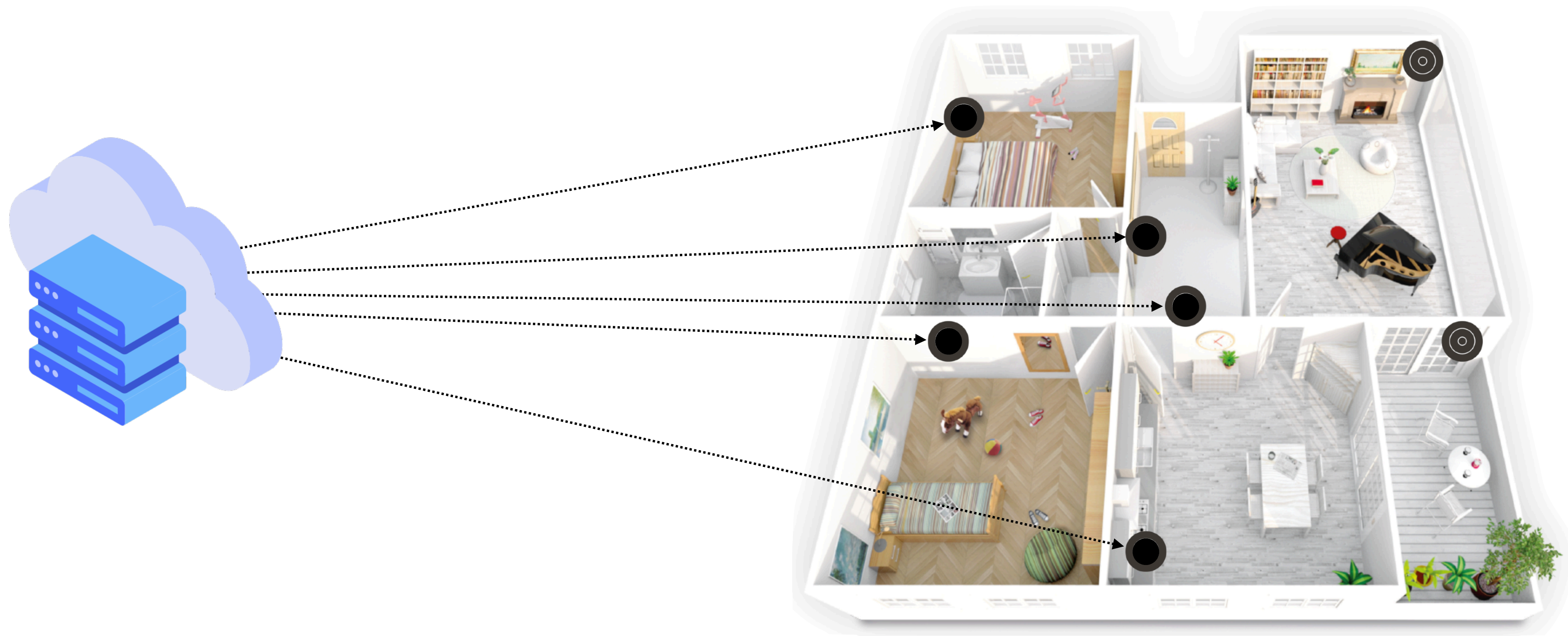
```
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();  
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

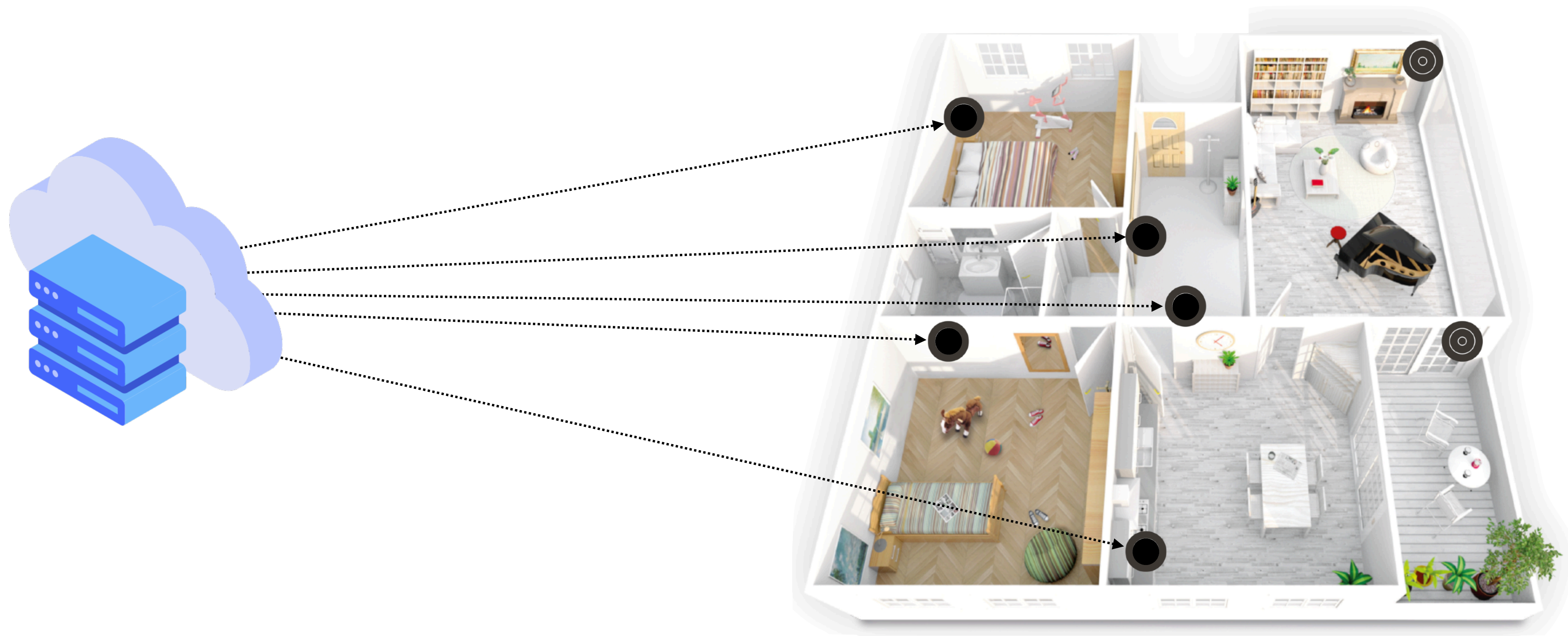
```
final ExecutorService executor = Executors.newCachedThreadPool();  
final List<Future<SensorData>> futures = executor.invokeAll(callables);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```
final List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
```

```
final List<Callable<SensorData>> callables = new ArrayList<>();  
for (SensorInfo sensorInfo : sensorInfos) { ... }
```

```
final ExecutorService executor = Executors.newCachedThreadPool();  
final List<Future<SensorData>> futures = executor.invokeAll(callables, 1, SECONDS);  
final List<SensorData> datum = new ArrayList<>(sensorInfos.size());  
for (Future<SensorData> future : futures) {  
    datum.add(future.get());  
}
```




```

List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();

```




```

List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();

```

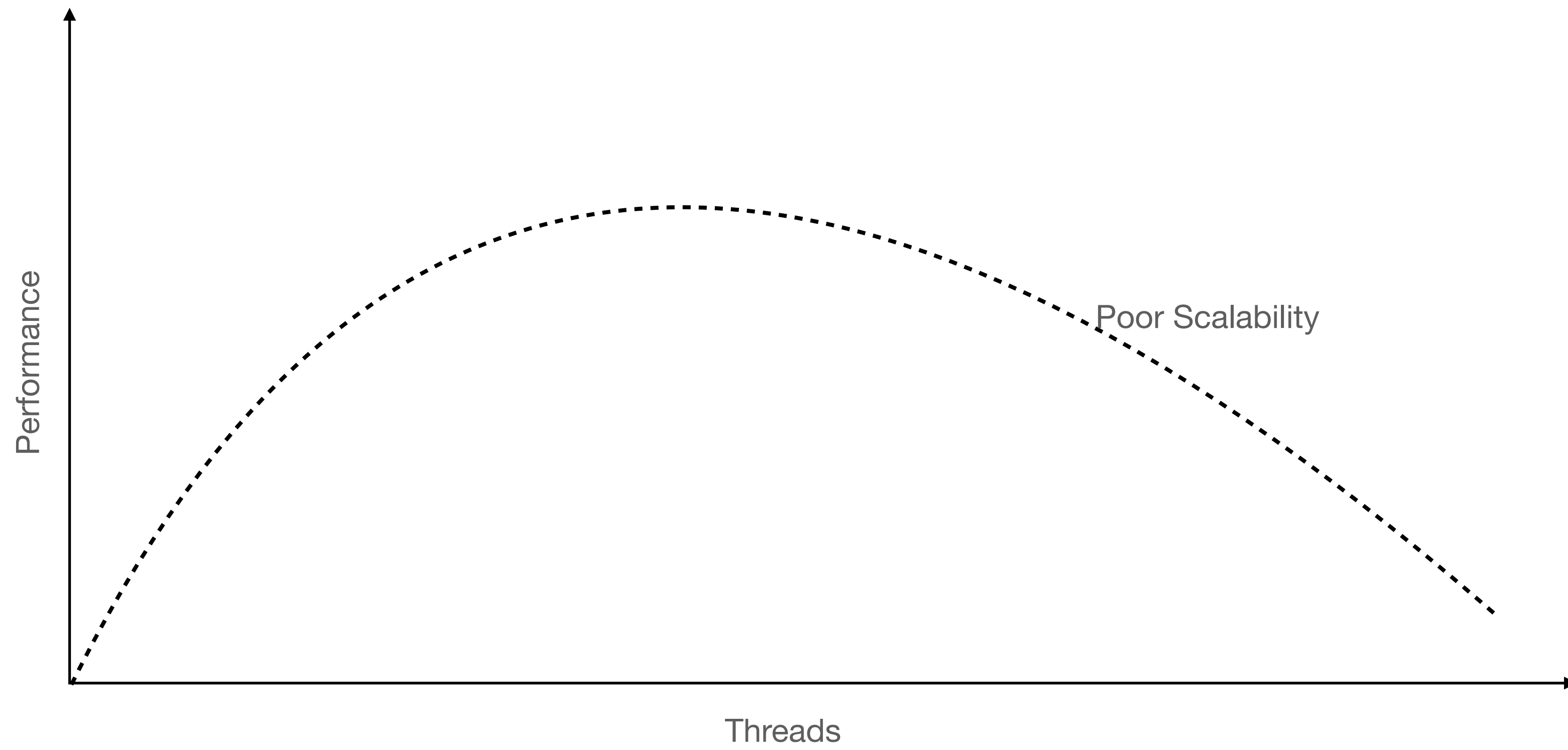



```

List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();

```





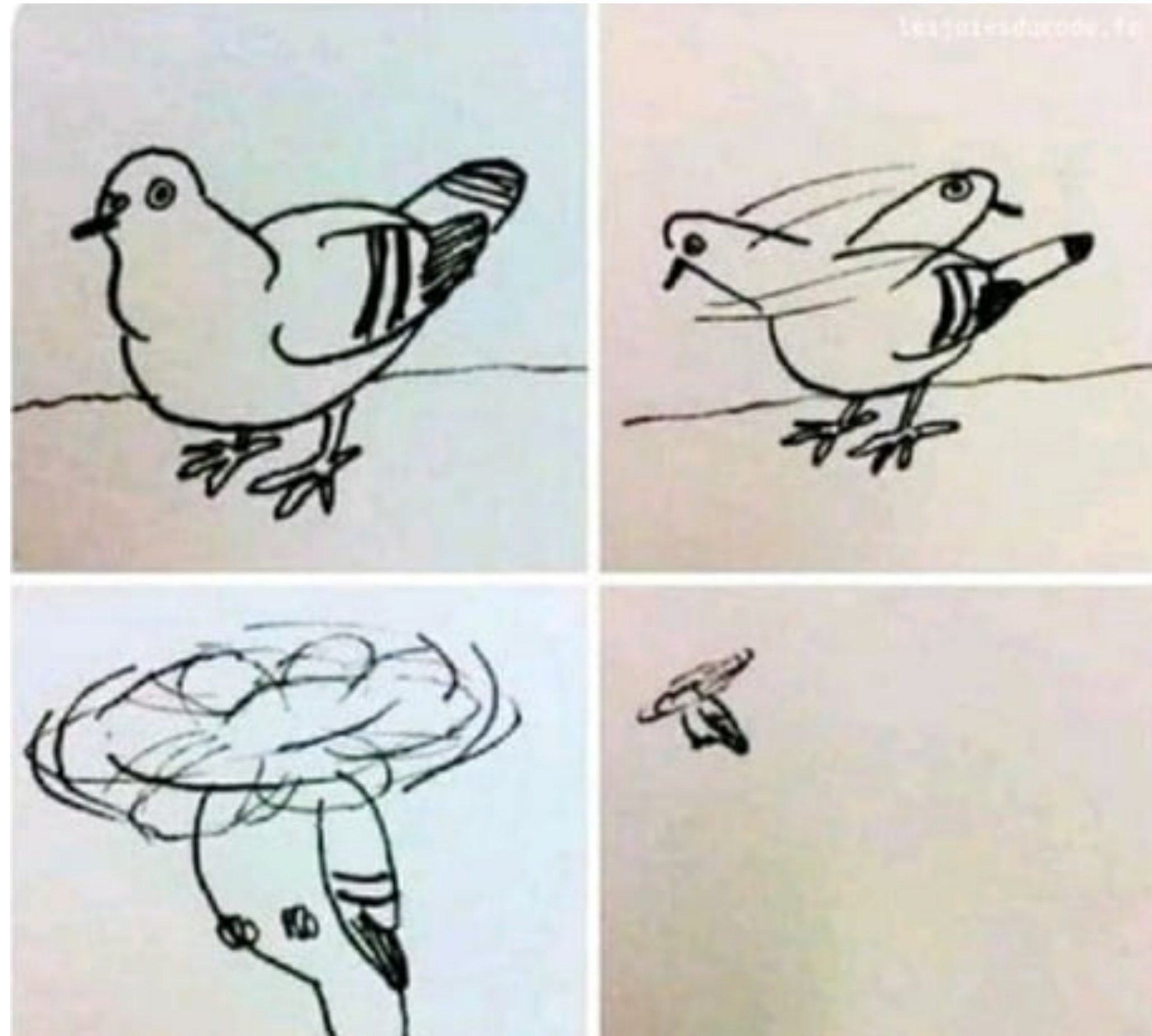
***Async* + Non-Blocking**

```
interface SensorClient {  
    CompletableFuture<Double> temperatureForSensorId(String id);  
}
```



```
interface SensorClient {  
    CompletableFuture<Double> temperatureForSensorId(String id);  
}
```

The code is mess even though it works



Reactive + Non-Blocking




```
interface SensorClient {  
    Mono<Double> temperatureForSensorId(String id);  
}
```

```
interface SensorClient {  
    Mono<Double> temperatureForSensorId(String id);  
}
```

Reactive Libraries API

> **Flux#flatMap(..)**

- merge **concurrently**



> **Flux#concatMap(..)**

- merge **sequentially**

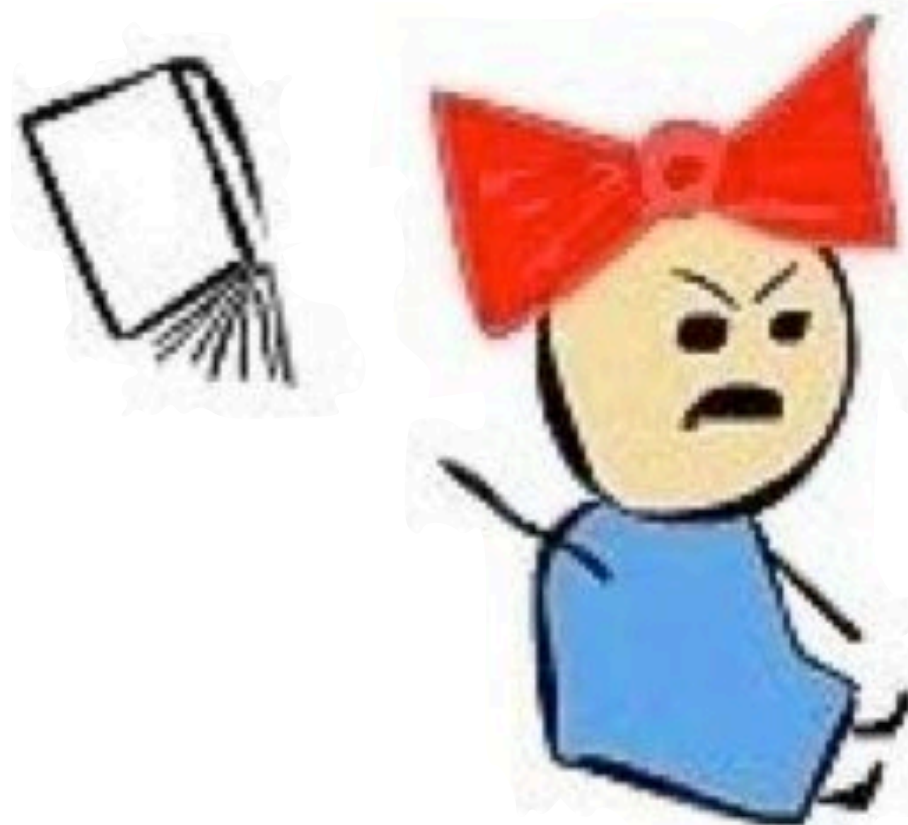


> **Mono#flatMap(..)**

- merge **sequentially**



ну ваш Reactive ...



.... буду писать на Imperative Java



Reactive Libraries API

> **Flux#flatMap(..)**

- merge **concurrently**



> **Flux#concatMap(..)**

- merge **sequentially**

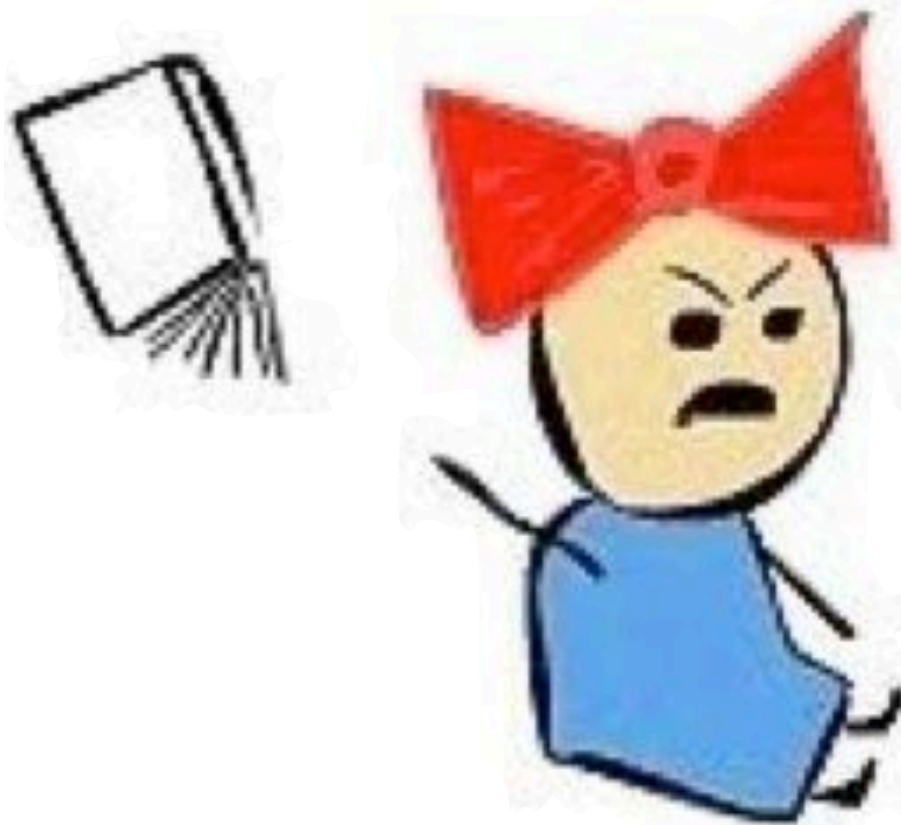


> **Mono#flatMap(..)**

- merge **sequentially**



ну ваш Reactive ...

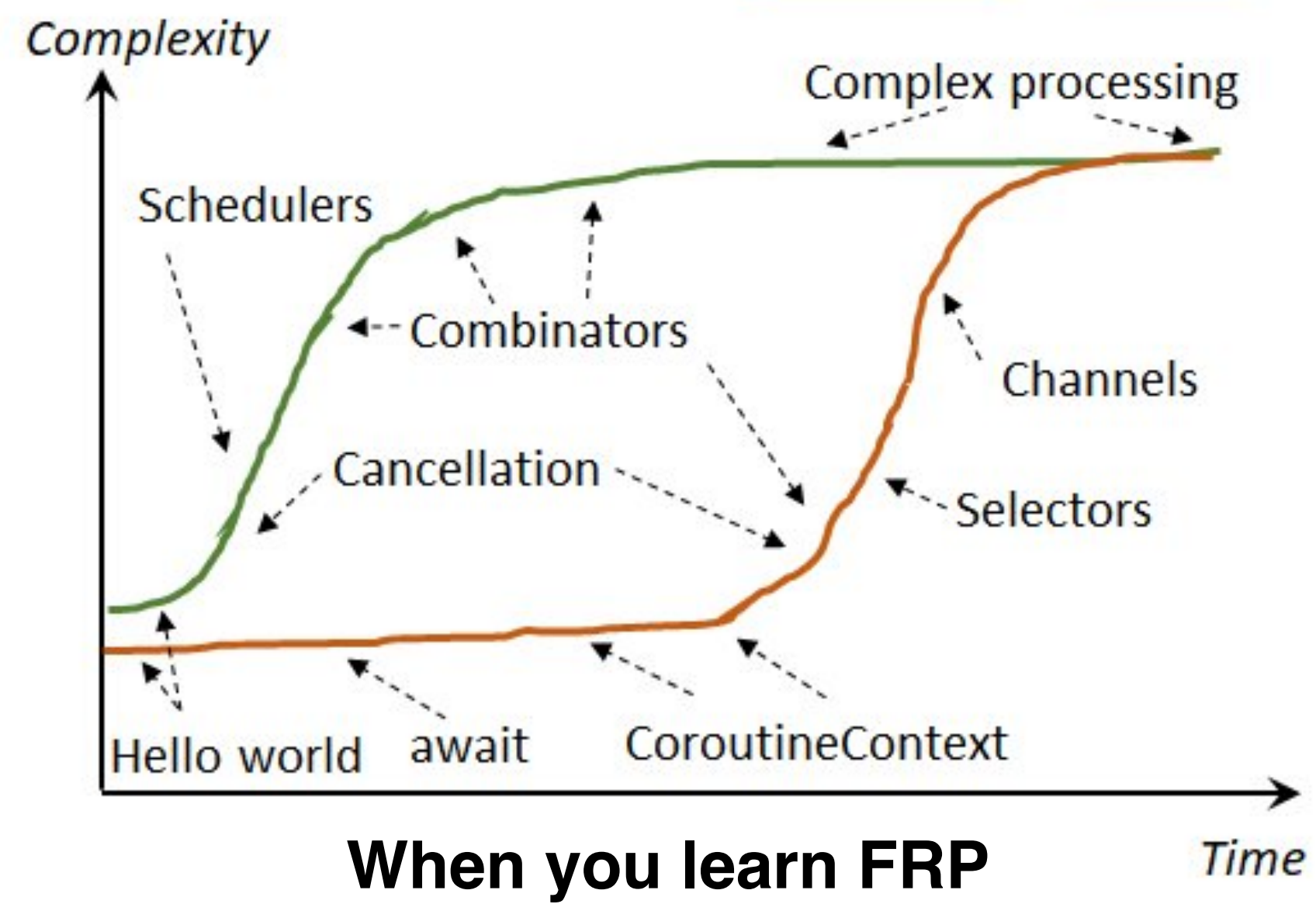


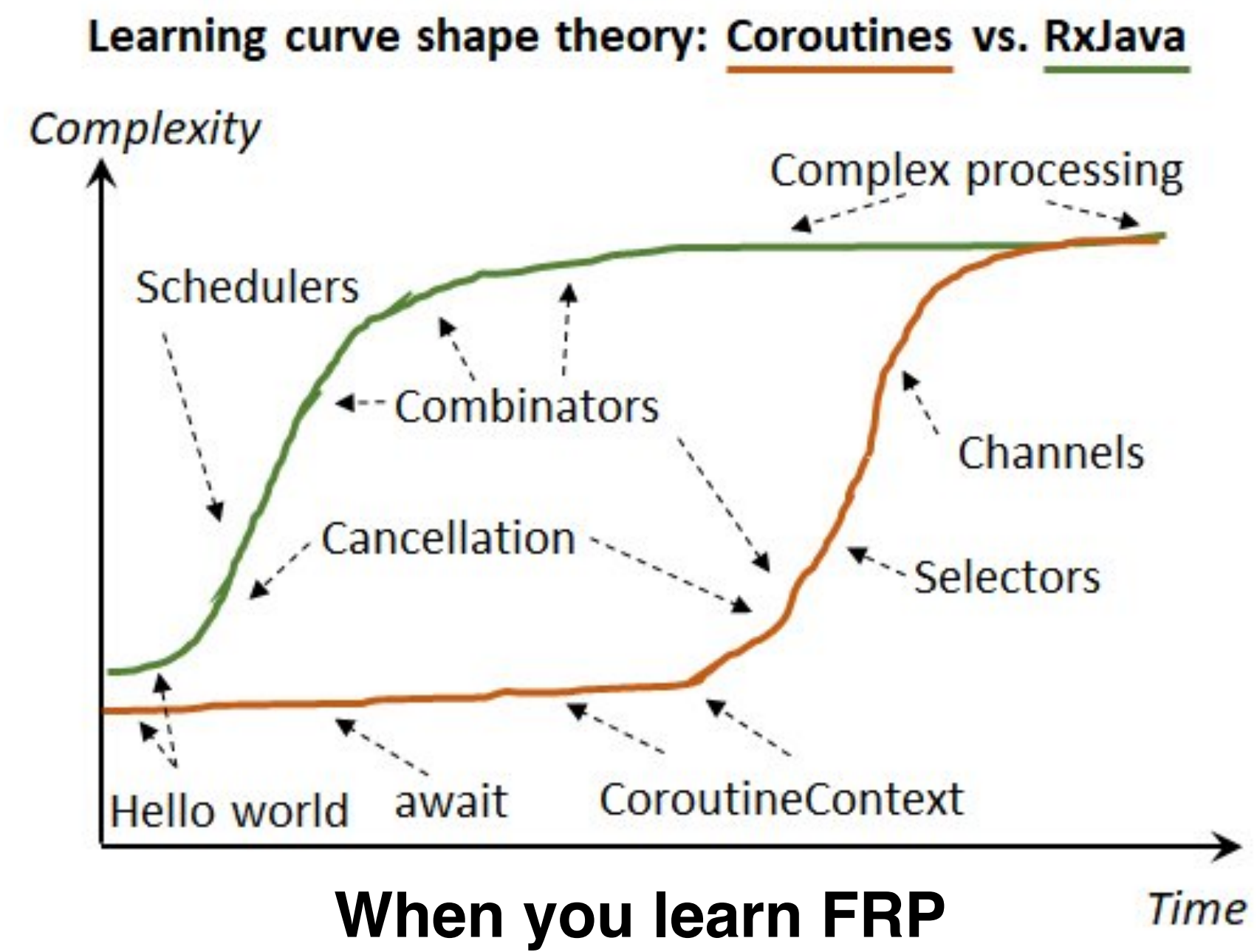
... заплачу еще 100k за Amazon и

.... буду писать на Imperative Java

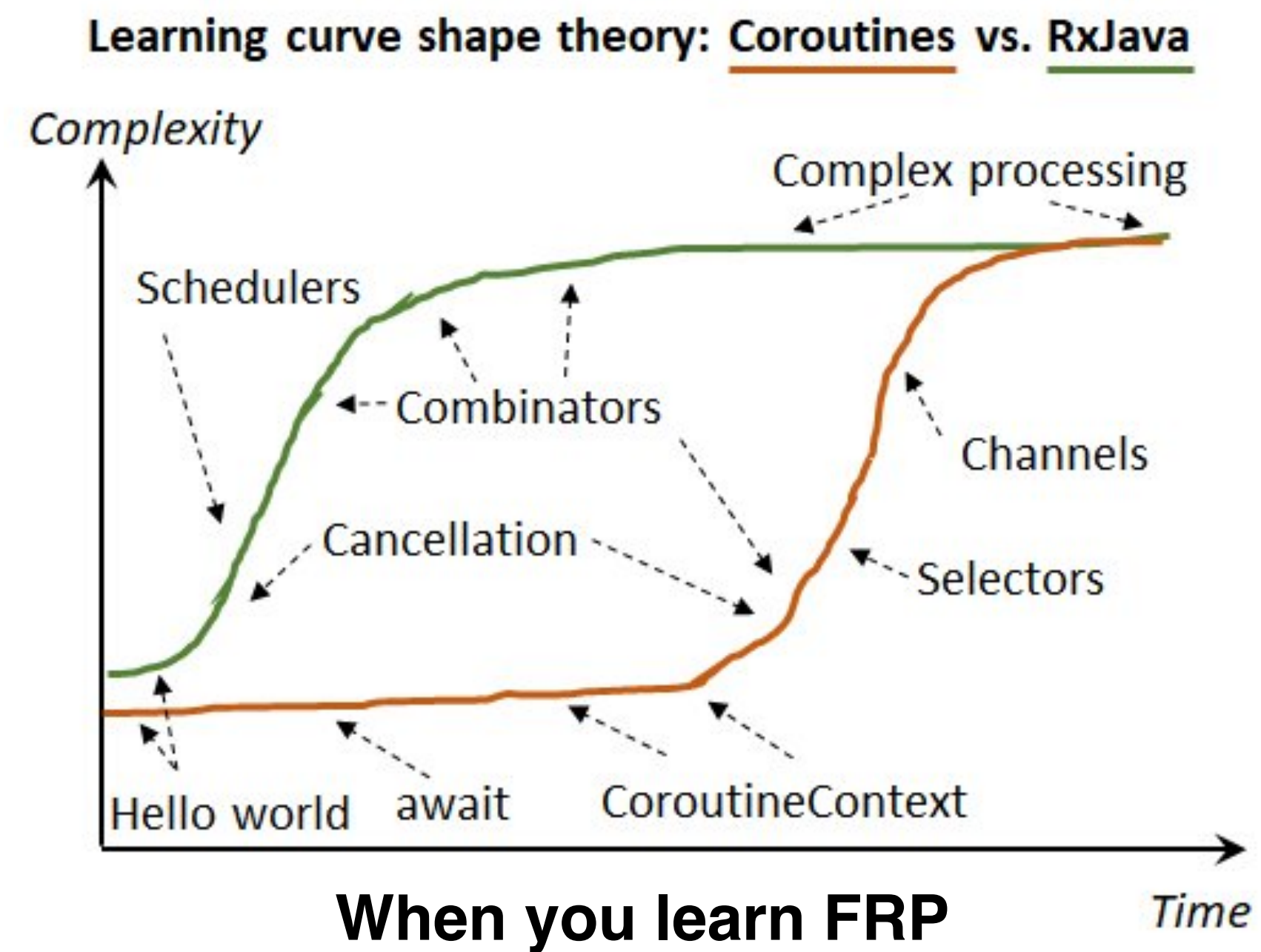


Learning curve shape theory: Coroutines vs. RxJava





When you look at someones FR code



When you look at someones FR code



When you debugging FR code

Learning curve shape theory: Coroutines vs. RxJava

Complexity

Complex processing

```
Exception in thread "main" reactor.core.Exceptions$ErrorCallbackNotImplemented: java.lang.RuntimeException: Boom!
Stack trace:
    at org.kds.reactive.debug.DebugDemo.lambda$debuggingUsingHook$2(DebugDemo.java:42)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:107)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)
    at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.tryOnNext(FluxDistinct.java:474)
    at reactor.core.publisher.FluxIterable$IterableSubscriptionConditional.fastPath(FluxIterable.java:585)
    at reactor.core.publisher.FluxIterable$IterableSubscriptionConditional.request(FluxIterable.java:470)
    at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.request(FluxDistinct.java:511)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)
    at reactor.core.publisher.LambdaSubscriber.onSubscribe(LambdaSubscriber.java:119)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)
    at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.onSubscribe(FluxDistinct.java:428)
    at reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:135)
    at reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:63)
    at reactor.core.publisher.Flux.subscribe(Flux.java:8134)
    at reactor.core.publisher.Flux.subscribeWith(Flux.java:8298)
    at reactor.core.publisher.Flux.subscribe(Flux.java:8105)
    at reactor.core.publisher.Flux.subscribe(Flux.java:8032)
    at reactor.core.publisher.Flux.subscribe(Flux.java:7975)
```



When you debugging FR code



When you look at someones FR code



When you seen an error from CompletionStage

Learning curve shape theory: Coroutines vs. RxJava

Complexity

Complex processing

```
Exception in thread "main" reactor.core.Exceptions$ErrorCallbackNotImplemented: java.lang.RuntimeException: Boom!  
Stack trace:
```

```
at org.kds.reactive.debug.DebugDemo.lambda$debuggingUsingHook$2(DebugDemo.java:42)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:107)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)  
at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.tryOnNext(FluxDistinct.java:474)  
at reactor.core.publisher.FluxIterable$IterableSubscriptionConditional.fastPath(FluxIterable.java:585)  
at reactor.core.publisher.FluxIterable$IterableSubscriptionConditional.request(FluxIterable.java:470)  
at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.request(FluxDistinct.java:511)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)  
at reactor.core.publisher.LambdaSubscriber.onSubscribe(LambdaSubscriber.java:119)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)  
at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)  
at reactor.core.publisher.FluxDistinct$DistinctFuseableSubscriber.onSubscribe(FluxDistinct.java:428)  
at reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:135)  
at reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:63)  
at reactor.core.publisher.Flux.subscribe(Flux.java:8134)  
at reactor.core.publisher.Flux.subscribeWith(Flux.java:8298)  
at reactor.core.publisher.Flux.subscribe(Flux.java:8105)  
at reactor.core.publisher.Flux.subscribe(Flux.java:8032)  
at reactor.core.publisher.Flux.subscribe(Flux.java:7975)
```



When you debugging FR code



When you look at someones FR code



When you seen an error from CompletionStage







Project Loom





Brian Goetz



Brian Goetz




```
interface SensorClient {  
    double temperatureForSensorId(String id);  
}
```

```
interface SensorClient {  
    double temperatureForSensorId(String id);  
}
```



```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```

```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```

```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```

```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
public List<SensorInfo> listUsersHouseSensors(String userId, String houseId) {
    final List<HouseInfo> houses = housesRepository.findById(userId);
    for (SensorInfo sensorInfo : sensorInfos) {
        for (HouseInfo house : houses) {
            if (houseId.equals(house.id())) {
                double t = sensorClient.temperatureForSensorId(sensorInfo.id());
                return sensorsRepository.findByIds(house.sensorsIds());
            }
            SensorData sensorData = new SensorData(...);
            datum.add(sensorData);
            latch.countDown();
        }
        throw new IllegalArgumentException(...);
    }
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```

Jetty and Loom

Greg Wilkins [gregw at webtide.com](mailto:gregw@webtide.com)

Wed Dec 30 08:14:55 UTC 2020

- Previous message (by thread): [building loom](#)
- **Messages sorted by:** [[date](#)] [[thread](#)] [[subject](#)] [[author](#)]

Hi,

The Jetty team have blogged about our initial experiments with Loom virtual threads:

- <https://webtide.com/do-looms-claims-stack-up-part-1/>
- <https://webtide.com/do-looms-claims-stack-up-part-2/>


```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```

```
double t = sensorClient.temperatureForSensorId(sensorInfo.id());
SensorData sensorData = new SensorData(...);
datum.add(sensorData);
latch.countDown();
});
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
double t = sensorClient.temperatureForSensorId(sensorInfo.id());
SensorData sensorData = new SensorData(...);
datum.add(sensorData);
latch.countDown();
});
}
● latch.await(1000, TimeUnit.MILLISECONDS);
  executor.shutdownNow();
```

```

double t = sensorClient.temperatureForSensorId(sensorInfo.id());
SensorData sensorData = new SensorData(...);
datum.add(sensorData);
latch.countDown();
});
}

```

● latch.await(1000, TimeUnit.MILLISECONDS);
 executor.shutdownNow();

Services

Debugger Console Actuator

Frames

Variables

✓ "<unnamed>"@6,003 in group "VirtualThreads": RUNNING

listUsersHouseSensorsTemperatures:54, SensorsService (com.example.demo. temperatures:29, SensorsController (com.example.demo.controllers)

invokeVirtual:-1, LambdaForm\$DMH/0x00000008010ec400 (java.lang.invoke)

invoke:-1, LambdaForm\$MH/0x00000008013a5800 (java.lang.invoke)

invokeExact_MT:-1, LambdaForm\$MH/0x00000008011d8800 (java.lang.invoke)

invoke:83, NewAccessorImplFactory\$1 (jdk.internal.reflect)

invoke:93, NativeMethodAccessorImpl (jdk.internal.reflect)

invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)

invoke:568, Method (java.lang.reflect)

doInvoke:205, InvocableHandlerMethod (org.springframework.web.method.su

invokeForRequest:150, InvocableHandlerMethod (org.springframework.web.m

invokeAndHandle:117, ServletInvocableHandlerMethod (org.springframework.v

invokeHandlerMethod:805, RequestMappingHandlerAdapter (org.springframe

Evaluate expression or add a watch

Thread.currentThread().isVirtual() = true

this = {SensorsService@6194}

userId = "1"

houseId = "1"

sensorInfos = {ImmutableCollections\$ListN@6190} size = 1

datum = {CopyOnWriteArrayList@6191} size = 0

executor = {ThreadPoolExecutor@6192}

latch = {CountDownLatch@6193} "java.util.concurrent.CountDownLatch@5bae009d[Count = 1]

sensorClient = {MockSensorClient@6195}

Switch frames from anywhere in the IDE with ⌘⇧↑ and ⌘⇧↓


```

double t = sensorClient.temperatureForSensorId(sensorInfo.id());
SensorData sensorData = new SensorData(...);
datum.add(sensorData);
latch.countDown();
});
}

```

● latch.await(1000, TimeUnit.MILLISECONDS);
 executor.shutdownNow();

Services

Debugger Console Actuator

Frames

✓ "<unnamed>"@6,003 in group "VirtualThreads": RUNNING

listUsersHouseSensorsTemperatures:54, SensorsService (com.example.demo.
 temperatures:29, SensorsController (com.example.demo.controllers)
 invokeVirtual:-1, LambdaForm\$DMH/0x00000008010ec400 (java.lang.invoke)
 invoke:-1, LambdaForm\$MH/0x00000008013a5800 (java.lang.invoke)
 invokeExact_MT:-1, LambdaForm\$MH/0x00000008011d8800 (java.lang.invoke)
 invoke:83, NewAccessorImplFactory\$1 (jdk.internal.reflect)
 invoke:93, NativeMethodAccessorImpl (jdk.internal.reflect)
 invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)
 invoke:568, Method (java.lang.reflect)
 doInvoke:205, InvocableHandlerMethod (org.springframework.web.method.su
 invokeForRequest:150, InvocableHandlerMethod (org.springframework.web.m
 invokeAndHandle:117, ServletInvocableHandlerMethod (org.springframework.v
 invokeHandlerMethod:805, RequestMappingHandlerAdapter (org.springframe

Variables

Evaluate expression or add a watch

Thread.currentThread().isVirtual() = true

this = {SensorsService@6194}
 p userId = "1"
 p houseId = "1"
 sensorInfos = {ImmutableCollections\$ListN@6190} size = 1
 datum = {CopyOnWriteArrayList@6191} size = 0
 executor = {ThreadPoolExecutor@6192}
 latch = {CountDownLatch@6193} "java.util.concurrent.CountDownLatch@5bae009d[Count = 1]
 sensorClient = {MockSensorClient@6195}

Switch frames from anywhere in the IDE with ⌘⇧↑ and ⌘⇧↓

```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperatureForSensorId(sensorInfo.id());
        SensorData sensorData = new SensorData(...);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
List<SensorInfo> sensorInfos = listUsersHouseSensors(userId, houseId);
List<SensorData> datum = new CopyOnWriteArrayList<>();
ExecutorService executor = Executors.newFixedThreadPool(sensorInfos.size());
CountDownLatch latch = new CountDownLatch(sensorInfos.size());
for (SensorInfo sensorInfo : sensorInfos) {
    executor.submit(() -> {
        double t = sensorClient.temperature(sensorInfo.id());
        SensorData sensorData = new SensorData(sensorInfo.id(), t);
        datum.add(sensorData);
        latch.countDown();
    });
}
latch.await(1000, TimeUnit.MILLISECONDS);
executor.shutdownNow();
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    return executor.invokeAll(callables, 1, SECONDS)
        .stream()
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    return executor.invokeAll(callables, 1, SECONDS)
        .stream()
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```

```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    Future<List<SensorData>> f = executor.submit(() ->
        executor.invokeAll(callables, 1, SECONDS)
            .stream()
            .filter(Future::isCompletedNormally)
            .map(Future::join)
            .toList());
    // do other actions here
    return f.join();
}
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
    Future<List<SensorData>> f = executor.submit(() ->
        executor.invokeAll(callables, 1, SECONDS)
            .stream()
            .filter(Future::isCompletedNormally)
            .map(Future::join)
            .toList());
    // do other actions here
    return f.join();
}
```

```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
```

```
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
```

```
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```

```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();

try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);

    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
    // do other actions here
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```

```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();
```

```
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
    // do other actions here
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```



```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();

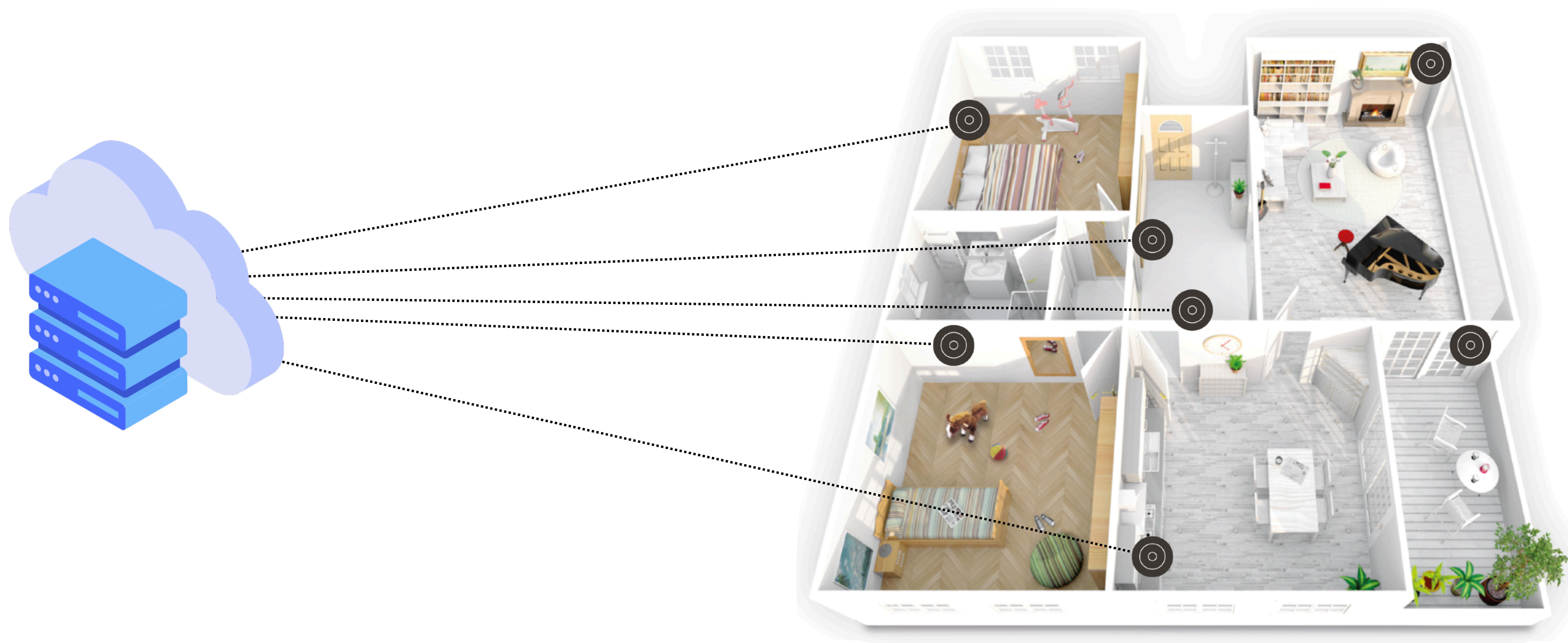
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
    // do other actions here
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```

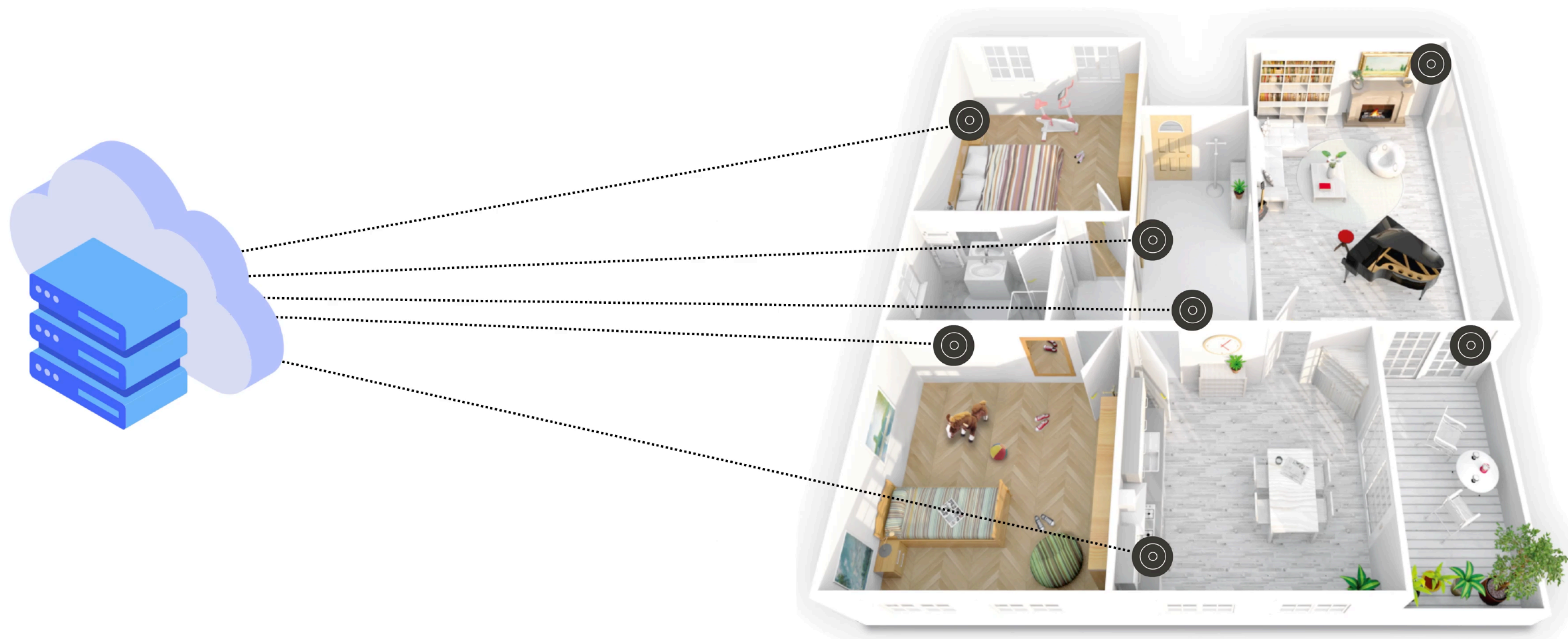
```
List<Callable<SensorData>> callables = listUsersHouseSensors(userId, houseId)
    .stream()      | SensorInfo
    .map(...)      | Callable<SensorData>
    .toList();

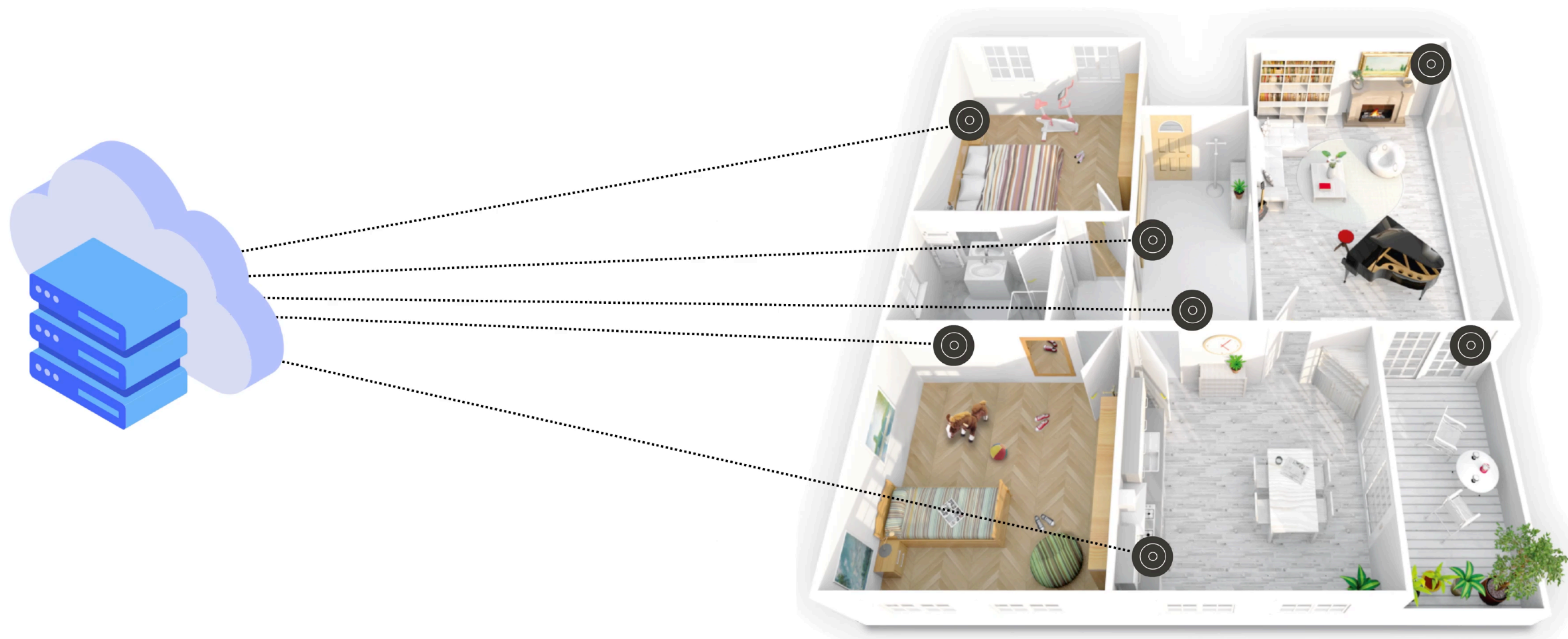
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    Stream<Future<SensorData>> futuresStream = executor.submit(callables);
    // do other actions here
    return futuresStream
        .filter(Future::isCompletedNormally)
        .map(Future::join)
        .toList();
}
```


Streaming









```
interface SensorClient {  
    void temperatureStreamForSensorId(String id,  
                                     Consumer<Double> c);  
}
```



```
interface SensorClient {  
    void temperatureStreamForSensorId(String id,  
        Consumer<Double> c);  
}
```

HET!

Observer Pattern




```
interface SensorClient {  
    Iterable<Double> temperatureStreamForSensorId(String id);  
}
```

```
interface SensorClient {  
    Iterable<Double> temperatureStreamForSensorId(String id);  
}
```


Reactive



Reactive $\neq$ Observer



1997 - Functional Reactive Animation

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in *Fran* are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. *Fran* has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia animations (involving audio, pictures, video, 2D and 3D graphics) has long been a complex and tedious job. Much of the difficulty, we believe, stems from the lack of sufficiently high-level abstractions, and in particular from the failure to clearly distinguish between *modeling* and *presentation*, or in other words, between *what* an animation is and *how* it should be presented. Consequently, the resulting programs must explicitly manage common implementation chores that have nothing to do with the content of an animation, but rather its presentation through low-level display libraries running on a sequential digital computer. These implementation chores include:

- stepping forward discretely in time for simulation and for frame generation, even though animation is conceptually continuous;

- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the “what” of an interactive animation, one can hope to then automate the “how” of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/-presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language properties for supporting modeled animations. For these reasons, *Fran* provides these data types in the programming language Haskell [9].

Advantages of Modeling over Presentation

The benefits of a modeling approach to animation are similar to those in favor of a functional (or other declarative) programming paradigm, and include clarity, ease of construction, composability, and clean semantics. But in addition there are application-specific advantages that are in some ways more compelling, painting the picture from a software engineering and end-user perspective. These advantages include the following:

- *Authoring.* Content creation systems naturally construct models, because the end users of such systems think in terms of models and typically have neither the expertise nor interest in programming presentation details.
- *Optimizability.* Model-based systems contain a presentation sub-system able to render any model that can be constructed within the system. Because higher-level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization.
- *Regulation.* The presentation sub-system can also more easily determine level-of-detail management, as well as sampling rates required for interactive animations, based on scene complexity, machine speed and load, etc.

To appear the International Conference on Functional Programming, June 1997, Amsterdam.



Reactive Programming combines:

- *Functional programming*
- *Observer pattern*
- *Iterable pattern*



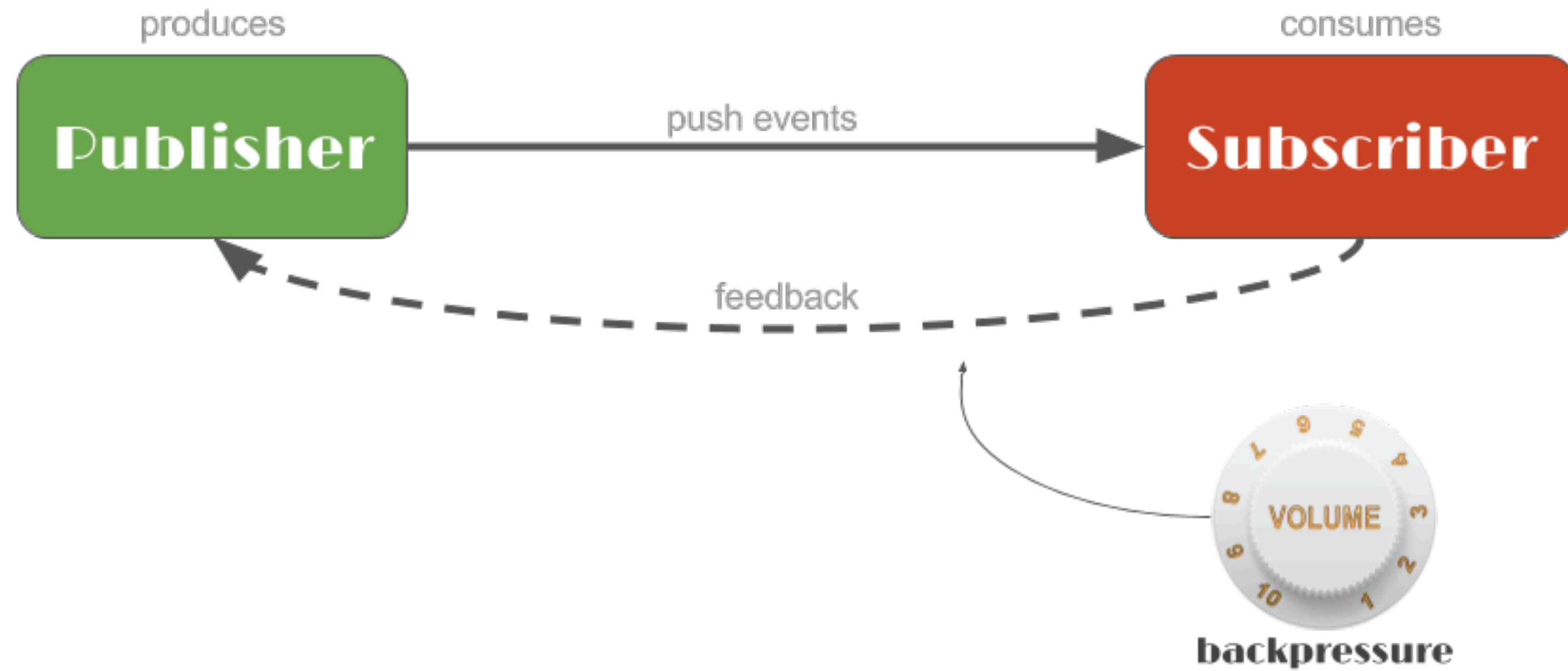
Reactive Programming is about programming with data streams

*Reactive Programming is about **programming with data streams***

Reactive Streams Specification



Reactive Streams Specification



JEP 266: More Concurrency Updates

<i>Owner</i>	Doug Lea
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Closed / Delivered
<i>Release</i>	9
<i>Component</i>	core-libs / java.util.concurrent
<i>Discussion</i>	core dash libs dash dev at openjdk dot java dot net
<i>Effort</i>	S
<i>Duration</i>	S
<i>Relates to</i>	JEP 193: Variable Handles
<i>Reviewed by</i>	Chris Hegarty, Martin Buchholz, Paul Sandoz
<i>Endorsed by</i>	Brian Goetz
<i>Created</i>	2015/08/04 11:01
<i>Updated</i>	2017/04/24 23:02
<i>Issue</i>	8132960

Summary

An interoperable publish-subscribe framework, enhancements to the CompletableFuture API, and various other improvements.

JEP 266: More Concurrency Updates

<i>Owner</i>	Doug Lea
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Closed / Delivered
<i>Release</i>	9
<i>Component</i>	core-libs / java.util.concurrent
<i>Discussion</i>	core dash libs dash dev at openjdk dot java dot net
<i>Effort</i>	S
<i>Duration</i>	S
<i>Relates to</i>	JEP 193: Variable Handles
<i>Reviewed by</i>	Chris Hegarty, Martin Buchholz, Paul Sandoz
<i>Endorsed by</i>	Brian Goetz
<i>Created</i>	2015/08/04 11:01
<i>Updated</i>	2017/04/24 23:02
<i>Issue</i>	8132960

Summary

An interoperable **publish-subscribe framework**, enhancements to the `CompletableFuture` API, and various other improvements.

Loom + Reactive



Better structured with Loom and Reactive

```
Flux<SensorInfo> listUsersHouseSensors(String userId, String houseId) { ... }
```

```
Flux<SensorData> streamUsersHousesSensorsTemperatures(String userId, String houseId) {  
    return listUsersHouseSensors(userId, houseId)  
        .flatMap(sensorInfo ->  
            sensorClient.temperatureStreamForSensorId(sensorInfo.id())  
                .map(t -> new SensorData(sensorsInfo, t, Instant.now()))  
        );  
}
```

Better structured with Loom and Reactive

```
List<SensorInfo> listUsersHouseSensors(String userId, String houseId) { ... }
```

```
Flux<SensorData> streamUsersHousesSensorsTemperatures(String userId, String houseId) {  
    return listUsersHouseSensors(userId, houseId)  
        .flatMap(sensorsInfo ->  
            sensorClient.temperatureStreamForSensorId(sensorsInfo.id())  
                .map(t -> new SensorData(sensorsInfo, t, Instant.now()))  
        );  
}
```

Better structured with Loom and Reactive

```
List<SensorInfo> listUsersHouseSensors(String userId, String houseId) { ... }
```

```
Flux<SensorData> streamUsersHousesSensorsTemperatures(String userId, String houseId) {  
    return Flux.fromIterable(listUsersHouseSensors(userId, houseId))  
        .flatMap(sensorsInfo ->  
            sensorClient.temperatureStreamForSensorId(sensorsInfo.id())  
                .map(t -> new SensorData(sensorsInfo, t, Instant.now()))  
        );  
}
```


Better structured with Loom and Reactive

```
List<SensorInfo> listUsersHouseSensors(String userId, String houseId) { ... }
```

```
Iterable<SensorData> streamUsersHousesSensorsTemperatures(String userId, String houseId) {  
    return Flux.fromIterable(listUsersHouseSensors(userId, houseId))  
        .flatMap(sensorsInfo ->  
            sensorClient.temperatureStreamForSensorId(sensorsInfo.id())  
                .map(t -> new SensorData(sensorsInfo, t, Instant.now()))  
        )  
        .toIterable();  
}
```

Better integration with Loom and Reactive

```
Mono.fromCallable() -> {  
    return blockingCall();  
}
```

Better integration with Loom and Reactive

```
Mono.fromCallable() -> {  
    return blockingCall();  
})  
.subscribeOn(Schedulers.boundedElastic())
```


Better integration with Loom and Reactive

```
Mono.fromCallable() -> {  
    return blockingCall();  
})  
.subscribeOn(Schedulers.fromExecutorService(Executors.newVirtualThreadExecutor()))
```

Better integration with Loom and Reactive

```
Mono.fromCallable() -> {  
    return blockingCall();  
})  
.subscribeOn(Schedulers.fromExecutorService(Executors.newVirtualThreadExecutor()))
```

Better integration with Loom and Reactive

```
monoSource
  .flatMap(value ->
    Mono.fromCallable() -> {
      return blockingCall(value);
    })
    .subscribeOn(Schedulers.fromExecutorService(Executors.newVirtualThreadExecutor()))
  )
```


Better integration with Loom and Reactive

```
monoSource  
  .map(value -> blockingCall(value))
```

Better integration with Loom and Reactive

monoSource

```
.publishOn(fromExecutorService(Executors.newVirtualThreadExecutor()))  
.map(value -> blockingCall(value))
```

Better generators with Loom and Reactive

```
Flux.create(emitter -> {  
    while(!emitter.isCancelled()) {  
        emitter.emit(...);  
    }  
})  
    .subscribeOn(Schedulers.virtualThreads())
```


Better generators with Loom and Reactive

```
Flux.create(emitter -> {  
    while(!emitter.isCancelled()) {  
        emitter.emit(...);                <- block Thread if there is no demand  
    }  
})  
.subscribeOn(Schedulers.virtualThreads())
```

Loom is friend of Reactive

Loom going to simplify apps

Loom going to simplify apps

```
List<SensorInfo> listUsersHouseSensors(  
    String userId,  
    String houseId) {  
    final List<HouseInfo> houses =  
        housesRepository.findByUserId(userId);  
  
    for (HouseInfo house : houses) {  
        if (houseId.equals(house.id())) {  
            return sensorsRepository.findByIds(house.sensorIds());  
        }  
    }  
}
```

Reactive for complex orchestration

```
listUsersHouseSensors(userId, houseId)
    .flatMap(sensorsInfo ->
        sensorClient.temperatureForSensorId(sensorsInfo.id())
            .map(t -> new SensorData(sensorsInfo, t, Instant.now()))
            .timeout(Duration.ofSeconds(1))
    );
```